View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

NASA Technical Memorandum 103961

TM-61
158518
P.17

Using a Cray Y-MP as an Array Processor for a RISC Workstation

Hugh LaMaster and Sarah J. Rogallo

(NASA-TM-103961) USING A CRAY Y-MP AS AN ARRAY PROCESSOR FOR A RISC WORKSTATION (NASA) 17 p

N93-25246

Unclas

G3/61 0158518

November 1992

National Aeronautics and Space Administration

NASA Technical Memorandum 103961

Using a Cray Y-MP as an Array Processor for a RISC Workstation

Hugh LaMaster and Sarah J. Rogallo, Ames Research Center, Moffett Field, California

November 1992

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

Summary

As microprocessors increase in power, the economics of centralized computing has changed dramatically. At the beginning of the 1980s, mainframes and supercomputers were often considered to be cost-effective machines for scalar computing. Today, microprocessor-based RISC (reduced-instruction-set computer) systems have displaced many uses of mainframes and supercomputers. Supercomputers are still cost competitive when processing jobs that require both large memory size and high memory bandwidth. One such application is array processing. Certain numerical operations are appropriate to use in a Remote Procedure Call (RPC)-based environment. Matrix multiplication is an example of an operation that can have a sufficient number of arithmetic operations to amortize the cost of an RPC call. This paper describes an experiment which demonstrates that matrix multiplication can be executed remotely on a large system to speed the execution over that experienced on a workstation.

Introduction

The title of this paper was chosen deliberately to be provocative. To many people familiar with traditional supercomputing, it sounds ludicrous. To others who are more familiar with distributed processing, it is a natural extension of current techniques to permit easier access to, and more efficient use of, unfamiliar and potentially hard-to-program supercomputers. This paper is a description of a reasonably successful attempt to split off the numerically intensive array-oriented operations from the largely scalar parts of application code.

The economic viability of current supercomputing practice is under assault by "killer micros." Microprocessor-based RISC (reduced-instruction-set computer) systems are now within a factor of four in speed on scalar code relative to far more expensive supercomputers. In some cases, the scalar speed of the supercomputer is roughly equal to that of the microprocessor-based system. Many programs are only partially vectorizable. A supercomputer user with a partially vectorizable, numerically intensive code is faced with a dilemma. The user can run the program locally on a workstation or server and wait a long time for an answer, or the user can run it on a supercomputer, such as the Cray Y-MP, and get the answer back faster but "waste" many far more valuable Cray CPU cycles. Supercomputers are designed to overcome the memory-bandwidth bottleneck that prevents large array-oriented programs from running fast on less expensive machines. But scalar code does not make use of that expensive memory bandwidth during most of its execution time. This situation motivated an experiment to develop an application in which the scalar and vector parts could be conveniently split apart.

The ideal for a distributed system is to run the scalar parts of a program on a low-cost workstation or server and to run the vector, or parallel, part on a suitable minisupercomputer, supercomputer, or massively parallel computer. Such computers are optimized for high speed on large-memory, numerically intensive tasks. With the current state of the art, it is very costly and time consuming to do this for individual applications. The cost of developing the code to split an existing application into scalar and vector parts is still very high. For this experiment, a different approach was chosen. A standard subroutine library, Level 3 Basic Linear Algebra Subprograms (BLAS3) (ref. 1), was chosen as the user interface. This is the natural choice, since many users already have calls to BLAS3 routines, in particular SGEMM, for which optimized versions exist on Cray, Convex, and other vector machines.

This paper describes a distributed application and the results of running it on various processors; the paper will show that it is feasible to split off large-array operations to another machine and perform the operations via RPC (Remote Procedure Call). Then, it will describe the limitations which apply to the current Cray system that restrict use of the technique to large problems.

The authors gratefully acknowledge the many constructive comments made by the reviewers.

1. Description of the Experiment

1.1 Motivation

The purpose of this experiment was to demonstrate the feasibility of developing a distributed application in which the compute-intensive part of the code could be conveniently separated from the scalar, or less compute-intensive, computations. The distributed application could then use network services to allow the partitioning of different parts of the application to separate and diverse computing resources that are best suited for the particular task.

This application demonstrates a distributed application that provides a user-level, FORTRAN-type subroutine call interface to the server process. The server was written to utilize a standard matrix multiplication subroutine, SGEMM. This subroutine was chosen because many users already have calls to SGEMM in their codes and because SGEMM is a standard routine for multiplying large arrays of data and is readily available in optimized form on many computer systems.

With the distributed-application approach used here, it is not necessary for the user to program the remote machine or convert any code. Instead, a library is linked to an existing program, and the computation takes place remotely with only limited involvement of the user.

1.2 Description of Application Design

The application created for this experiment is divided into two logical pieces, a client process and a server process. The client process contains the scalar code and initiates requests to the server process to perform some action (in this case, the compute-intensive process of matrix multiplication). The server process waits for a request to be made by the client and then performs the requested action. The client and server processes may run on the same computer system, or the server may run remotely on an independent system that is connected by a communication network to the client system.

The remote-procedure call model is similar to the usual local-procedure call model in that a single thread of control logically winds through the client (caller) process and through the server process. The caller process sends a message to the server process and waits for a reply. The message contains the parameters to be passed to the remote procedure. The reply contains the procedure's results. When the reply is received, the results are made available to the caller and the caller's execution is resumed.

The server is normally waiting for the arrival of a message from a caller. When a message arrives, the server process extracts the parameters for the remote procedure, calls a dispatch routine, performs the requested service, sends back a reply to the caller, and then waits for the next message.

To create the client/server interface code, networking services developed by Sun Microsystems were used (refs. 2 and 3). The Sun Remote Procedure Call (RPC) facility is a library of procedures that implement the logical client-to-server communications to support network applications. Sun RPC was used because it is implemented on a variety of operating systems, including Cray Unicos, SGI Irix, SunOS, Convex OS, and DEC Ultrix (Ultrix was lacking the RPC protocol compiler *rpcgen*).

The details of programming applications to use RPC can be tedious. One of the more difficult areas is passing data in a portable format between different computer architectures. The eXternal Data Representation (XDR) is a standard for the description and encoding of data. XDR uses a language to describe data formats in a concise manner. XDR library routines may be used to encode data from the local machine representation into a standard machine-independent format. The machine that receives the data uses XDR to decode the data from the standard representation to its own internal format. XDR relies heavily on the IEEE standard for floating-point data representation and is implemented most efficiently on architectures that use the IEEE standard.

The use of RPC procedures and the writing of XDR routines to convert procedure arguments and results into XDR network format, and vice versa, are facilitated by using *rpcgen*. *rpcgen* accepts a remote program interface definition written in the RPC language. It produces a C-language output for RPC programs. The output includes skeleton versions of the client and the server routines, XDR routines to handle both the parameters and results of the procedure, and a header file that contains common definitions. The client and server skeletons contain calls to RPC library procedures to handle the network communication. The developer writes the server procedure and links it with the server skeleton to produce an executable server program. The user (client) program that makes local procedure calls is linked with the client skeleton.

1.3 Application Code

The protocol specification for our remote program is written in the RPC language and defines the names of the procedures, the data types of their input parameters and output results, the name of the remote program, and its version name. Each procedure name, program name, and version name is assigned a number. A remote procedure is uniquely identified by its program number, version number, and procedure number.

The code labeled `proto.x` (listed in the Appendix) shows the protocol specification for our remote program that contains one version, and has six procedure definitions. The program name is `REMOTE_MATRIX_PROG`. The program number is selected arbitrarily from the range of numbers defined by the RPC protocol for user-defined services. The version name is `REMOTE_MATRIX_VERS` (with version no. 1). Six procedure names and numbers are specified with definitions of their input parameters and output results. For example, "`int R_SMATRIXALLOC (int) = 1;`" specifies a procedure named `R_SMATRIXALLOC` with an integer input parameter and an integer output result (the procedure number is 1). "`singlep_matrix R_SGEMM (sgemm_args) = 5;`" specifies a procedure named `R_SGEMM` (with procedure no. 5). The procedure's input parameters are defined by the previously declared structure `sgemm_args`. `sgemm_args` contains the parameters necessary for calling the matrix multiplication routine `SGEMM`. The remote-procedure output results are defined by the structure `singlep_matrix`. `singlep_matrix` declares an array of floats that will contain the results of the matrix multiplication.

The server program is designed to utilize the BLAS3 routines `SGEMM`, single-precision matrix multiplication, and `DGEMM`, double-precision matrix multiplication. The server program contains six remote procedures, two of which allocate space for the single- or double-precision result matrix, two of which release the allocated space, and one

each to call SGEMM and DGEMM. The code labeled `server.c` (listed in the Appendix) shows the six procedures that are linked with the server skeleton code (generated by *rpcgen*) to produce the running server program. `server.c` contains the procedures that do the "real work" of the server program, such as allocating space for the result matrix, calling the matrix multiplication routine SGEMM or DGEMM, and freeing the allocated memory after the result matrix has been returned to the client. `server.c` contains the code that corresponds to the six procedures defined in `proto.x`. The procedure-naming convention is to use the remote-procedure name declared in the prototype definition (e.g., R_SGEMM), convert it to lower-case letters, append an underline ("_"), and add the version number (here 1) to produce the name `r_sgemm_1`. Note that remote procedures always use a pointer to their arguments and a pointer to their results since these data will be processed by the appropriate XDR routines.

SGEMM and DGEMM perform the matrix-matrix operation $C = \alpha AB + \beta C$ where α and β are scalars and A, B, and C are matrices. They also perform the same basic operation with either matrix A or B transposed.

The code labeled `sgemm.c` (listed in the Appendix) shows one of the client-side procedures (there is another one for using DGEMM) that is linked with the client skeleton code (generated by *rpcgen*) to produce the user-callable interface to the remote procedure server. `sgemm.c` defines the standard user interface to SGEMM. The first part of the code is concerned with checking the validity of the input parameters. If the input parameters are valid, the procedure uses an environment variable to obtain the name of the system on which the remote server is running. Then, a data structure known as a "client handle" is created to be passed to the skeleton client routines that will call the remote procedure. The skeleton routines are called the same way as the remote procedures are defined in `server.c` except that the client handle is inserted as a second argument. After executing a call to a remote procedure, the result pointer is checked to determine if there was an error resulting from a failure of the RPC mechanism. If not, the results are available for use by the caller.

1.4 Network and Operating System Environment

The main network at Ames Research Center (ARC) is referred to as ARCLAN. ARCLAN is largely Ethernet based. Ethernet physical media are coaxial cable, optical fiber, unshielded twisted pair, and some microwave links. Link level routers, network level routers, and other hosts and IP gateways provide connectivity over the campus. Some systems are connected via Fiber Distributed Data Interface (FDDI), although most systems are connected directly to Ethernet. There are also additional networks using UltraNet and Hyperchannel, although they are not part of ARCLAN and were not used for this experiment.

Although a faster medium than Ethernet would have improved effectiveness, Ethernet was fast enough to carry out the described experiment.

2. Performance Results

2.1 Server Times on Various Machines

The results shown in table 1 are the total user time consumed by the server. This time includes XDR overhead and the time for the multiplication of two square matrices using SGEMM. In some cases, the machines were idle. In others (Convex, Cray) the machines were fully loaded and 100% busy. These results are not intended to be an accurate benchmark of these machines under controlled conditions. The purpose is to characterize the conditions under which the computation can be effectively distributed. Several versions of SGEMM were tried in order to get the best possible times. The rows noted "orig" in table 1 are the original SGEMM in BLAS3. The rows denoted "mod" are for a version modified by the authors to be cache-contained in small, nonvector, RISC workstations. The "lib" version is the one supplied by the system vendor, if available. For the Cray, this is the Cray-supplied, multitasked, parallel version. All client matrix elements are 32 bits. All server matrix multiplies are 32 bits, except the Cray Y-MP matrix multiply, which is 64 bits, although the RPC is done in 32 bits.

2.2 Effectiveness Comparisons

The previous results tell only part of the story. In order to compare the effectiveness of distributed computation versus local computation, and the effectiveness of distributed computation on different systems, several cases will be scrutinized more closely. A sample N = 800 case was compared between two machines in order to examine the overhead.

Based on an analysis of current production and research codes being run at ARC, and considering the relative costs and capabilities of RISC workstations and the Cray, a goal of achieving 50 MFLOPS on the Cray was established for this experiment.

For the N = 800 case, the following data were gathered for the situation where the local workstation was a Sun 4/75 and the remote server was on a Cray Y-MP8/864:

Elapsed time on server: 64.4 seconds
Total CPU time on server (including system time):
26.4 seconds
User CPU time on server: 20.9 seconds

Table 1. User CPU seconds for an N × N matrix multiply including XDR overhead

Array dimension N -----> machine --------v	100	200	400	600	800	1000
Sun 4/490 - orig	0.9	7.0	55.4	185.4	436.6	846.9
Sun 4/490 - mod	0.4	3.2	25.6	80.5	198.1	388.0
Sun 4/75 - orig	0.7	5.5	42.4	140.5	332.3	645.7
Sun 4/75 - mod	0.4	2.9	21.4	70.2	165.8	321.2
SGI 4D/30 - orig	0.6	4.4	34.3	112.4	265.7	514.2
SGI 4D/30 - mod	0.3	2.8	22.6	73.4	175.8	334.4
DEC DS/5500 - orig	0.4	4.4	33.1	110.9	261.5	512.5
DEC DS/5500 - mod	0.4	2.8	22.9	76.7	186.6	366.8
IBM 320 - orig	0.6	3.4	21.1	63.8	145.2	276.8
IBM 320 - mod	0.5	2.7	17.7	52.4	121.9	229.0
IBM 320 - lib	0.4	1.7	8.8	23.5	48.9	86.9
Convex C210 - mod	0.8	3.9	19.1	51.3	106.6	N/A
Convex C210 - orig	0.8	3.7	18.0	47.7	98.6	N/A
Convex C210 - lib	0.8	3.2	14.4	35.0	68.6	N/A
Cray Y-MP8/864 - lib	0.3	1.2	5.1	12.0	20.9	35.7

User CPU time on server to do only the multiply:
3.45 seconds

Rate on server while doing only the multiply:
296.8 MFLOPS
Rate on server (user time only): 49.0 MFLOPS
User-visible rate of server over network: 15.9 MFLOPS

User CPU time to do the multiply on local machine:
157.7 seconds
Rate on local machine while doing the multiply:
6.5 MFLOPS

In this case, accessing the remote server achieved an elapsed time speedup of 2.4 over doing the calculation locally on the Sun 4/75. This is less than the 4X speedup, which is generally considered necessary to interest a user in a significantly different computing method. For smaller arrays, the benefits are less, or even negative. However, on the plus side, the rate on the Cray is close to the established goal of 50 MFLOPS.

On further analysis, the Cray is unusual when compared with other machines: the XDR overhead is relatively much greater. For example, the XDR conversion of the input matrices costs 5.1 seconds on the Sun 4/75, while on the Cray, the time is 10.5 seconds. When the approximate 4X scalar speedup of the Cray is considered, it appears that the Cray is about eight times slower doing data "conversion." No doubt this is due to the Cray's non-IEEE floating-point format. The Convex supports IEEE in hardware, but had the same problem as the Cray. This is because the Convex was lacking IEEE RPC libraries.

Therefore, while the method is effective, the RPC/XDR overhead on the Cray and Convex is far greater than it should be, and the realized benefit is significantly less than it should be. It is unknown at this time whether significant improvements are possible in the efficiency of the Cray XDR routines, but clearly the most desirable approach from the standpoint of distributed computation is for Cray to support IEEE floating-point arithmetic in hardware and software.

2.3 Which Operations Are Potential Candidates?

Dense arrays have order $O(N^2)$ elements where N is the dimension of the array. Calculations that are suitable candidates have > $O(N^2)$ operations. For example, matrix

multiply using SGEMM, used in this paper, has $O(N^3)$ operations. We will examine this case using the number of CPU clock ticks per operation. Assume that the number of clock ticks per floating-point operation is roughly constant. We will denote clock ticks per operation as K_1. Also assume that the system overhead of transmitting and converting an element is roughly constant. This constant, in clock ticks per element, is denoted K_2 below. Note that SGEMM uses three input matrices and one output matrix, requiring the transmission via RPC of four matrices. SGEMM actually requires $2N^3+3N^2$ arithmetic operations. Ignoring the N^2 portion of the work, the ratio of work performed to overhead is approximately:

$$\frac{2K_1N^3}{4K_2N^2} = \frac{1}{2}\left(\frac{K_1}{K_2}\right)N$$

When this ratio is large enough, the operation will be feasible. The measured speed for SGEMM on the Cray when N = 800 is about 297 MFLOPS. To achieve the goal of 50 MFLOPS for the remote server, no more than about 5/6 the total work can be overhead. So, the above ratio can be no more than 1/6. By measurement on the Cray, K_1 is about 0.5618 clock ticks per operation, and K_2 is about 1134 clock ticks per element, considering user time only. Considering both user time and system time, K_2 is about 1497. Hence, N must be approximately 897 in order to achieve the 50-MFLOPS goal. This is a large matrix for the workstation to handle locally, and would take several hundred seconds on most current workstations.

If the Cray RPC/XDR conversion were about eight times faster, as would be expected if the Cray were to use IEEE floating-point arithmetic, then K_2 would drop to about 708 (including system overhead), and the feasible size would drop to N = 424. This assumes that the "system" time is due to network activity that is independent of floating-point format. However, if the non-XDR user and system overhead (assumed to be due to network overhead) could be reduced to be more in line with the network overhead of a typical workstation, then considering the relative scalar speeds of the Cray and workstation, the Cray network overhead would be about 1/3 of its current value. Then K_2 would drop to 311, and the feasible matrix size would drop to about 186. This size matrix could easily be handled locally on a workstation. A production client/server implementation would normally be written to do the multiplication of small matrices locally.

Table 2 and figure 1 show the observed MFLOPS for various cases with the remote server running on a Sun 4/75 and on a Cray Y-MP8/864. The version of SGEMM optimized for workstations by the authors was used on the Sun, and the vendor-supplied library version was used on the Cray.

The interesting thing about these timing results is that RPC overhead is small in the case of the Sun; there is little change in the overall efficiency over the range of array sizes. Conversely, on the Cray, there is a great deal of RPC overhead, and consequently there is a large increase in performance as array sizes increase. It is inefficient to remotely process small arrays on the Cray because of the large RPC overhead. On the other hand, there is a large performance advantage to be gained by using the Cray on large, dense arrays. Figure 1 illustrates this.

Returning to the question of which operations are economic, we are looking for functions, such as matrix multiply, in which the Operation/Transmission ratio is greater than $O(1)$. For example, matrix multiplication and standard LU decomposition for dense matrices require $O(N^3)$ operations and require transmission of $O(N^2)$ elements. In contrast, matrix vector operations require $O(N^2)$ operations and transmission of $O(N^2)$ elements. Hence, matrix vector operations are not normally suitable for this type of distributed application. Other candidates include Fast Fourier Transforms (FFTs), since an FFT requires transmission of $O(N)$ elements, while requiring $O(N\log_2 N)$ operations. However, for a single FFT, it is unlikely that the overhead would be low enough because $\log_2 N$ grows very slowly with N. Computational chemistry is an area with excellent prospects for distributed applications since some algorithms are $O(N^4)$, or even $O(N^5)$ or greater (ref. 4).

3. CONCLUSIONS

The experiment demonstrated that using a distributed client-server approach between a workstation and a

Table 2. Comparison of Cray Y-MP and Sun workstation user MFLOPS

Array size ----------> machine---v	100	200	400	600	800	1000
Sun 4/75 - mod	5.0	5.52	5.98	6.15	6.18	6.23
Cray Y-MP8/864 - lib	6.7	13.33	25.1	36.00	46.54	56.02

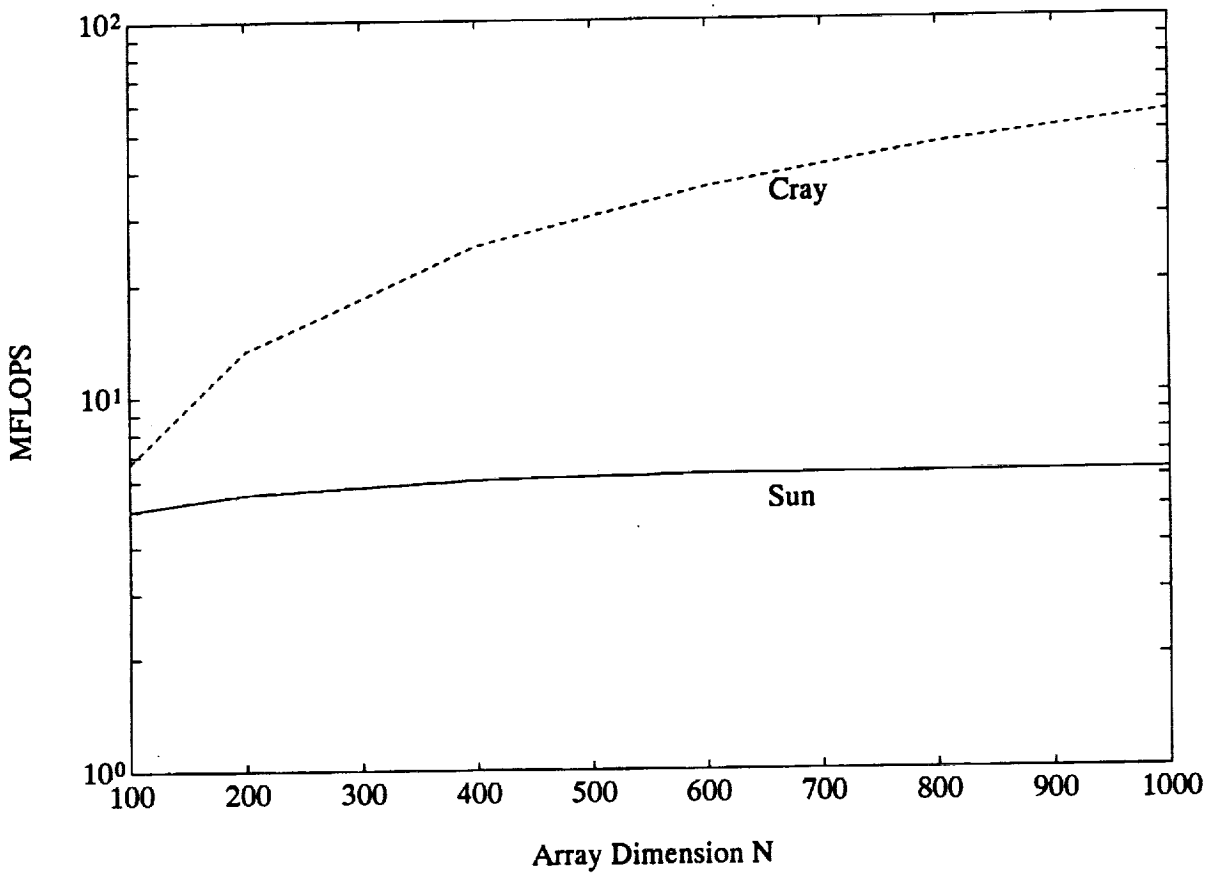

Figure 1. MFLOPS vs N for Cray and Sun servers.

supercomputer can work today on some array-oriented operations, such as matrix multiply, when applied to large, dense problems. The inefficiencies of translating between IEEE and Cray native format restrict the method to large arrays of order N = 900 or greater. If the supercomputer in question were to use IEEE floating-point format and had networking efficiency comparable to the best performance available today in workstations, then the client-server approach could be practical for much smaller problems, and also a wider variety of problems. It is likely, however, that for the foreseeable future, only functions in which the ratio of the number of operations to the number of elements is at least *O(N)* will be economic.

Finally, the results suggest that it might make sense in some cases to dedicate a special-purpose computing node in a network to function as an array-processor server for client machines in the same network. This could be the ideal use for a massively parallel machine that might otherwise be difficult to use efficiently in a general way with current compilers. With the distributed-application approach used here, it would not be necessary for the user to program the remote machine or convert any code. Instead, a library could be developed such that the user could link the library to an existing program, and then the remote computation would take place transparently to the user.

REFERENCES

1. Dongarra, J. J.; Du Croz, J.; Hammarling, S.; and Duff, I.: A Set of Level 3 Basic Linear Algebra Subprograms. ACM Trans. Math. Softw., vol. 16, no. 1, 1990, pp. 1–17.

2. Bloomer, J.: Power Programming with RPC O'Reilly & Associates, Inc., Sebastopol, Ca., 1991.

3. Network Programming Guide, Sun Microsystems, Inc., Rev. A 27, March 1990.

4. Arnold, J. O.: Computational Chemistry. Supercomputing in Aerospace (conference proceedings), Paul Kutler, ed., NASA CP-2454, March 1987, pp. 299–305.

APPENDIX

proto.x

```
/*****************************************************************************
*                                                                            *
* proto.x --  remote matrix procedure protocol description                   *
*                                                                            *
*****************************************************************************/

struct sgemm_args {
        char  transa;
        char  transb;
        int   m;
        int   n;
        int   k;
        float alpha;
        float a<>;                  /*  input matrix a                    */
        int   lda;
        float b<>;                  /*  input matrix b                    */
        int   ldb;
        float beta;
        float c<>;                  /*  input matrix c                    */
        int   ldc;
        };

struct singlep_matrix {
        float c<>;                  /*  output matrix c                   */
        };

struct dgemm_args {
        char  transa;
        char  transb;
        int   m;
        int   n;
        int   k;
        double alpha;
        double a<>;                 /*  input matrix a                    */
        int   lda;
        double b<>;                 /*  input matrix b                    */
        int   ldb;
        double beta;
        double c<>;                 /*  input matrix c                    */
        int   ldc;
        };

struct doublep_matrix {
        double c<>;                 /*  output matrix c                   */
        };

                                    /*  program definition                */
program REMOTE_MATRIX_PROG {
        version REMOTE_MATRIX_VERS {
                int R_SMATRIXALLOC ( int ) = 1;
                int R_SMATRIXFREE ( int ) =  2;
                int R_DMATRIXALLOC ( int ) = 3;
                int R_DMATRIXFREE ( int ) =  4;
                singlep_matrix R_SGEMM ( sgemm_args ) = 5;
                doublep_matrix R_DGEMM ( dgemm_args ) = 6;
        } = 1;  /*  version number  */
} = 536870912;  /*  program number 0x20000000  */
```

server.c

```
/*****************************************************************************
*                                                                            *
*  server.c  -  remote procedure that calls Level 3 BLAS                     *
*                                                                            *
*****************************************************************************/

#include <rpc/rpc.h>
#include "proto.h"                                  /*  generated by rpcgen  */

#ifndef _CRAY
#  define SGEMM sgemm_                             /*  fix-up function names  */
#  define DGEMM dgemm_
#endif
#ifdef _CRAY
#  define DGEMM SGEMM
#endif

static float *smatrix;              /*  ptr to a single precision matrix     */
static int ssize;                   /*  number of elements in smatrix        */
static double *dmatrix;             /*  ptr to a double precision matrix     */
static int dsize;                   /*  number of elements in dmatrix        */

/******** Allocate memory for a single precision result matrix ***************/
int *
r_smatrixalloc_1 ( dimension )
     int *dimension;
{
  static int status;                                    /*  Must be static!  */

  ssize = *dimension;
  smatrix = (float *)malloc( (unsigned)(ssize * sizeof(float)) );
  if( smatrix == 0 )
    status = 0;  /*  not O.K.  */
  else
    status = 1;  /*  O.K.  */

  return(&status);
}

/******** Allocate memory for a double precision result matrix ***************/
int *
r_dmatrixalloc_1 ( dimension )
     int *dimension;
{
  static int status;                                    /*  Must be static!  */

  dsize = *dimension;
  dmatrix = (double *)malloc( (unsigned)(dsize * sizeof(double)) );
  if( dmatrix == 0 )
    status = 0;  /*  not O.K.  */
  else
    status = 1;  /*  O.K.  */

  return(&status);
}

/***** Free allocated memory for a single precision result matrix ************/
```

```
int *
r_smatrixfree_1 ( argp )
     int *argp;
{
  static int status;                                    /*  Must be static!  */

  free( (char *)smatrix );
  status = 1;  /*  O.K.  */

  return(&status);
}

/***** Free allocated memory for a double precision result matrix ************/
int *
r_dmatrixfree_1 ( argp )
     int *argp;
{
  static int status;                                    /*  Must be static!  */

  free( (char *)dmatrix );
  status = 1;  /*  O.K.  */

  return(&status);
}

/****************************** Call SGEMM  ***********************************/
struct singlep_matrix *
r_sgemm_1 ( args )
     struct sgemm_args *args;
{
  static struct singlep_matrix out;                     /*  Must be static!  */

  int i;

  for( i = 0; i < ssize; i++ )  smatrix[i] = args->c.c_val[i];

  SGEMM( &args->transa, &args->transb,  &args->m,  &args->n,  &args->k,
         &args->alpha,   args->a.a_val, &args->lda,
                         args->b.b_val, &args->ldb,
         &args->beta,    smatrix,       &args->ldc );

  out.c.c_len = ssize;
  out.c.c_val = smatrix;

  return(&out);
}

/****************************** Call DGEMM  ***********************************/
struct doublep_matrix *
r_dgemm_1 ( args )
     struct dgemm_args *args;
{
  static struct doublep_matrix out;                     /*  Must be static!  */

  int i;

  for( i = 0; i < dsize; i++ )  dmatrix[i] = args->c.c_val[i];

  DGEMM( &args->transa, &args->transb,  &args->m,  &args->n,  &args->k,
```

```
        &args->alpha,    args->a.a_val, &args->lda,
                         args->b.b_val, &args->ldb,
        &args->beta,     dmatrix,       &args->ldc );

  out.c.c_len = dsize;
  out.c.c_val = dmatrix;

  return(&out);
}
```

sgemm.c

```
/*****************************************************************************

sgemm.c

sgemm_  -  procedure to do remote matrix multiplication using SGEMM
           from Level 3 BLAS.

   C := alpha*op( A ) * op( B ) + beta*C,

where   op( X ) is one of
   op( X ) = X  or op( X ) = X',
and
   alpha and beta are scalars, and A, B, and  C are matrices, with op( A )
   an m by k matrix,  op( B ) a k by n matrix, and C an m by n matrix.

*****************************************************************************/

#include <stdio.h>
#include <rpc/rpc.h>
#include "proto.h"

#ifndef _CRAY
#   define XERBLA xerbla_
#endif
#ifdef _CRAY
#   define sgemm_ SGEMM
#endif

#ifdef ultrix
#   include <time.h>
#endif
#ifdef sgi
#   include <sys/time.h>
#endif

#define MAX(a,b) (((a)>(b))?(a):(b))
#define MIN(a,b) (((a)<(b))?(a):(b))

void sgemm_ ( transa, transb, m, n, k, alpha, a, lda, b, ldb, beta, c, ldc )

char *transa; /*addr of form option for matrix op(A)                        */
char *transb; /*addr of form option for matrix op(B)                        */
int *m;       /*addr of number of rows of op(A) and C                       */
int *n;       /*addr of number of columns in op(B) and C                    */
int *k;       /*addr of number of columns of op(A) and rows of op(B)        */
float *alpha; /*addr of scalar alpha                                        */
float *a;     /*addr of array A
                           A(lda, k) if transa='N' or 'n'
                           A(lda, m) if transa != 'N' or 'n'                */
int *lda;     /*addr of first dimension of A                                */
float *b;     /*addr of array B
                           B(ldb, n) if transb='N' or 'n'
                           B(ldb, k) if transb != 'N' or 'n'                */
int *ldb;     /*addr of first dimension of B                                */
float *beta;  /*addr of scalar beta                                         */
float *c;     /*addr of array C
                           C(ldc, n)                                        */
```

```
int *ldc;     /*addr of first dimension of C                                  */

{
  static CLIENT *cl = NULL;     /*  client handle                              */
  static char *mptr = NULL;     /*  pointer to env. variable "MATRIX_SERVER"   */
  static char machine[256];     /*  name of server machine                     */
  static struct timeval tv = { 0, 0 };/*  structure to define timeout value   */
  static char proc_name[7] = "SGEMM ";

  int i;
  int nota, notb, nrowa, nrowb;
  int *status;
  int dummy;
  int csize;
  struct sgemm_args args;
  struct singlep_matrix *matrix_out;

  char *getenv();
  CLIENT *clnt_create();
  void timeproc();

/******************  Check input parameters ********************************/
  if( (strncmp(transa,"N",1)==0) || (strncmp(transa,"n",1)==0) )
    nota = TRUE;
  else
    nota = FALSE;

  if( (strncmp(transb,"N",1)==0) || (strncmp(transb,"n",1)==0) )
    notb = TRUE;
  else
    notb = FALSE;

  if( nota==TRUE ) nrowa = *m;
  else             nrowa = *k;

  if( notb==TRUE ) nrowb = *k;
  else             nrowb = *n;

  i = 0;
  if     ( !nota
        && (strncmp(transa,"T",1)!=0) && (strncmp(transa,"t",1)!=0)
        && (strncmp(transa,"C",1)!=0) && (strncmp(transa,"c",1)!=0) )
    i = 1;
  else if ( !notb
         && (strncmp(transb,"T",1)!=0) && (strncmp(transb,"t",1)!=0)
         && (strncmp(transb,"C",1)!=0) && (strncmp(transb,"c",1)!=0) )
    i = 2;
  else if ( *m < 0 )
    i= 3;
  else if ( *n < 0 )
    i = 4;
  else if ( *k < 0 )
    i = 5;
  else if ( *lda < MAX(1,nrowa) )
    i = 8;
  else if ( *ldb < MAX(1,nrowb) )
    i = 10;
  else if ( *ldc < MAX(1,*m) )
    i = 13;
```

```
  if ( i != 0 ) {
    XERBLA ( proc_name, &i );      /*  call the Level 2 BLAS error routine  */
    return;
  }

/*********************  Quick return if possible.  ***************************/
  if( ( *m==0 ) || ( *n==0 ) ||
      ( ( ( *alpha==0.0 ) || ( *k==0 ) ) && ( *beta==1.0 ) ) ) return;

/******************  Get the remote server machine  ***************************/
  mptr = getenv( "MATRIX_SERVER" );
  if( mptr != NULL ) {
    strcpy( machine, mptr );
  }
  else {
    fprintf(stderr, "Error: environment variable MATRIX_SERVER is not set to\
 the name of the server machine.\n");
    exit(1);
  }

/*****  Create client handle for calling REMOTE_MATRIX_PROG using "tcp"  *****/
  if( cl == NULL ) {
    cl = clnt_create( machine, REMOTE_MATRIX_PROG, REMOTE_MATRIX_VERS,
                      "tcp" );
    if( cl == NULL ) {             /*  Couldn't establish server connection  */
      clnt_pcreateerror( machine );
      exit(1);
    }
  }

/**************  Allocate space in server for result matrix:  *****************/
  tv.tv_sec = 25;                                  /*  timeout in seconds  */
  clnt_control( cl, CLSET_TIMEOUT, &tv );      /*  set client timeout value  */

  csize = *ldc * *n ;
  status = r_smatrixalloc_1( &csize, cl );     /*  allocate space in server  */
    if ( status == NULL ) {
      clnt_perror(cl, "RPC error from calling r_smatrixalloc_1");
      exit(1);
    }
  if ( *status == 0 ) {
    fprintf(stderr,
     "Error: Could not allocate memory for result matrix on %s.\n",
      machine);
    exit(1);
  }

/**************  Call remote procedure to multiply matrices:  *****************/
  tv.tv_sec = 2 * (25. + 1.E-6 * (2. * *m * *k * *n));    /* timeout in secs */
  clnt_control( cl, CLSET_TIMEOUT, &tv );      /*  set client timeout value  */

  args.transa = *transa;
  args.transb = *transb;
  args.m      = *m;
  args.n      = *n;
  args.k      = *k;
  args.alpha  = *alpha;
  if ( nota == TRUE )
```

```
    args.a.a_len = *lda * *k;  /*  op( A ) = A   */
  else
    args.a.a_len = *lda * *m;  /*  op( A ) = A'  */
  args.a.a_val = a;
  args.lda     = *lda;
  if ( notb == TRUE )
    args.b.b_len = *ldb * *n;  /*  op( B ) = B   */
  else
    args.b.b_len = *ldb * *k;  /*  op( B ) = B'  */
  args.b.b_val = b;
  args.ldb     = *ldb;
  args.beta    = *beta;
  args.c.c_len = csize;
  args.c.c_val = c;
  args.ldc     = *ldc;

  matrix_out = r_sgemm_1( &args, cl );                /*  multiply matrices  */
  if( matrix_out == NULL ) {
    clnt_perror( cl, "RPC error from calling r_sgemm_1" );
    exit(1);
  }
  for( i = 0; i < matrix_out->c.c_len; i++ ) {
    c[i] = matrix_out->c.c_val[i];
  }

/*********************  Free allocated server memory: ************************/
  tv.tv_sec = 25;                                     /*  timeout in seconds  */
  clnt_control( cl, CLSET_TIMEOUT, &tv );      /*  set client timeout value  */

  status = r_smatrixfree_1( &dummy, cl );             /*  free allocated space  */
  if( *status == NULL ) {
    clnt_perror(cl, "RPC error from calling r_smatrixfree_1");
    exit(1);
  }

  return;
}
```

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1992	Technical Memorandum

4. TITLE AND SUBTITLE

Using a Cray Y-MP as an Array Processor for a RISC Workstation

5. FUNDING NUMBERS

505-59

6. AUTHOR(S)

Hugh LaMaster and Sarah J. Rogallo

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Ames Research Center
Moffett Field, CA 94035-1000

8. PERFORMING ORGANIZATION REPORT NUMBER

A-92158

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA TM-103961

11. SUPPLEMENTARY NOTES

Point of Contact: Hugh LaMaster, Ames Research Center, MS 233-9, Moffett Field, CA 94035-1000; (415) 604-1056

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified — Unlimited
Subject Category 61

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 words)***

As microprocessors increase in power, the economics of centralized computing has changed dramatically. At the beginning of the 1980s, mainframes and supercomputers were often considered to be cost-effective machines for scalar computing. Today, microprocessor-based RISC (reduced-instruction-set computer) systems have displaced many uses of mainframes and supercomputers. Supercomputers are still cost competitive when processing jobs that require both large memory size and high memory bandwidth. One such application is array processing. Certain numerical operations are appropriate to use in a Remote Procedure Call (RPC)-based environment. Matrix multiplication is an example of an operation that can have a sufficient number of arithmetic operations to amortize the cost of an RPC call. This paper describes an experiment which demonstrates that matrix multiplication can be executed remotely on a large system to speed the execution over that experienced on a workstation.

14. SUBJECT TERMS

Distributed systems, Distributed applications, BLAS3, SGEMM, Matrix multiply, RPC

15. NUMBER OF PAGES

14

16. PRICE CODE

A02

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified		